NASA/CR-2000-209851
ICASE Report No. 2000-4

Towards a Customizable PVS

Gerald Lüttgen and César Muñoz
ICASE, Hampton, Virginia

Ricky Butler, Ben Di Vito, and Paul Miner
NASA Langley Research Center, Hampton, Virginia

Institute for Computer Applications in Science and Engineering
NASA Langley Research Center
Hampton, VA

Operated by Universities Space Research Association

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Contract NAS1-97046

January 2000

TOWARDS A CUSTOMIZABLE PVS*

GERALD LÜTTGEN†, CÉSAR MUÑOZ†, RICKY BUTLER‡, BEN DI VITO‡, AND PAUL MINER‡

Abstract. PVS is a state-of-the-art theorem-proving tool developed by SRI International. It is used in a variety of academic and real-world applications by NASA and ICASE researchers, for whom tool customization and extensibility are becoming increasingly important issues. This paper shows, by referring to past experiences with several projects and case studies, that the customization features currently offered by PVS are often insufficient. It also suggests several improvements regarding PVS's customization in the short run and regarding its extensibility in the long run.

Key words. customization, extensibility, formal methods, PVS, theorem proving

Subject classification. Computer Science

1. Introduction. PVS is a general verification system developed and maintained by the Formal Methods Group at SRI International [32, 38, 39]. It combines a very rich specification language with a powerful, interactive theorem prover. The specification language of PVS is based on a classical, but typed, higher-order logic. The theorem prover integrates decision procedures for several kinds of theories and also allows one to incorporate user-defined proof strategies to automate the proof process as far as possible. NASA Langley [8, 30] is a long-time user of PVS, whose experiences with the theorem prover show that it is a well-performing tool, provided that the application under consideration can be tailored to PVS's problem solving style. Unfortunately, tailoring applications is difficult in practice, leading to a desire for tool customization and extensibility. This desire is shared by researchers at ICASE [19], whose main interest is the development of new verification technologies, especially heterogeneous techniques combining theorem proving, model checking, and type checking.

Customization and *extensibility* issues within formal specification and verification tools, such as PVS, are becoming increasingly important topics in Formal Methods research and technology transfer. The reason is that only tools with a high degree of flexibility and automation can cope with the complexity of today's digital systems and with the usability requirements imposed by hardware and software engineers. The main arguments for powerful customization and extensibility within PVS concern aspects of tool *integration* as well as tool *specialization*. A tight integration of PVS with external tools and environments would enable the use of specialized decision procedures within PVS, such as for model checking various temporal and modal logics or for reasoning about regular languages. Vice versa, other formal and informal tools for the design and analysis of digital systems could profit from PVS's elegant specification language and from its theorem-

*This work was supported by the National Aeronautics and Space Administration under NASA Contract No. NAS1-97046 while the first two authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681-2199, USA. This paper was presented at the PVS User Group Meeting held in conjunction with the World Congress on Formal Methods in the Development of Computing Systems (FM '99) in Toulouse, France, September 1999.

†Institute for Computer Applications in Science and Engineering (ICASE), Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-2199, USA, e-mail: {luettgen, munoz}@icase.edu.

‡Assessment Technology Branch, Mail Stop 130, NASA Langley Research Center, Hampton, VA 23681-2199, USA, e-mail: {r.w.butler, b.l.divito, p.s.miner}@larc.nasa.gov.

proving capabilities. However, in order to encourage engineers to apply formal specification and verification techniques, verification tools must become an integral part of engineering environments, such as UML tools in software design and VHDL tools in hardware design [7, 21]. Beside this aspect of tool integration, another desirable feature of verification tools is their specialization to particular problem domains. Naturally, this breaks down to two issues: (i) the specialization of the prover language in order to allow the interfacing of different specification languages to the verification tool under consideration, and (ii) the specialization of the prover itself, e.g., via user-defined proof strategies.

This paper first gives an overview of the current customization features in PVS. By referring to four case studies conducted by the authors, it is shown that these features are often insufficient to tailor PVS to some interesting academic and real-life applications. In addition, matters regarding extensibility seem to have been largely ignored during the development of PVS. After a detailed analysis of the underlying issues, this paper then develops several suggestions on how PVS's customization features can be significantly improved in the short term. These suggestions concern the mechanism and language for writing proof strategies, as well as syntactic and semantic aspects of the PVS language. This paper also elaborates on a vision for a next-generation PVS which asks for tool extensibility, and, thereby, carries the idea of tool customization one step further. Please note that some of the observations and suggestions made in this paper have also been reported by other researchers, and that the developers of PVS at SRI International are partially aware of them for quite some time.

The remainder of this paper is organized as follows. The next section gives an overview of the verification system PVS and its key components. It also surveys the customization features supported in PVS today. Section 3 illustrates our past experiences with customization issues in PVS, while Section 4 presents and discusses several suggestions on how to improve customization. The final section contains some remarks on tool extensibility and our concluding thoughts.

2. PVS: An Overview. The abbreviation PVS stood originally for *Prototype Verification System*, as the tool was conceived as a prototype for research on formal verification technology. The design of PVS was shaped by experience with the development of specification languages and theorem provers in the late Seventies and in the Eighties. In particular, PVS borrows from an earlier system developed at SRI International, the EHDM theorem prover [23, 42]. PVS is implemented in Lisp [44] using the Common Lisp Object System [10] and was first publicly released in 1992; the most current release is Version 2.3. Over the years, PVS matured from a prototype to a robust and powerful formal specification and verification system. It is used by many researchers and engineers in academic and industrial sites to attack complex problems in a broad spectrum of application domains [38], including software systems [14], hardware systems [13], and embedded systems [12]. Aspects under investigation ranged from safety criticality [11], over fault tolerance [26], to human-computer interaction [22]. The architecture of PVS, schematized in Figure 2.1, reflects that PVS is mainly used as an *interactive system*. The interface to the system has a textual and command-line form and is built on top of GNU's editor *Emacs* [43]. At this level, the user writes formal specifications that are then interactively analyzed either by the *type checker* and the *proof checker*, or animated using the *ground evaluator*.

Formal specifications are structured in so-called *theories*, which correspond to modules in programming languages [34]. During the initialization of PVS, a special, predefined theory, which is referred to as *prelude* and includes basic definitions, axioms, and propositions, is automatically loaded. A theory is an arrangement of declarations of mathematical and logical objects, such as types, (higher-order) functions, axioms, and

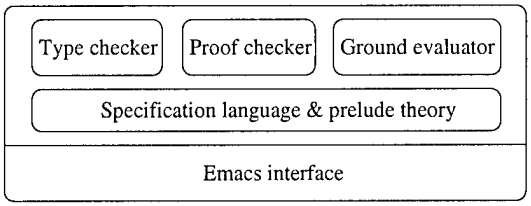

FIG. 2.1. *Schematized architecture of PVS*

theorems. Theories can be parameterized by mathematical objects, and they can in turn be imported by other theories. The specification language of PVS is based on classic *higher-order logic*, i.e., functions are first-class objects and quantification over general objects is supported. However, the language is enriched with an expressive *type system* and also supports operators known from *functional programming languages* [25], such as conditionals, local declarations, first-class functions, as well as record and function overriding.

The PVS language is strongly typed, i.e., objects need to be explicitly declared with their types [34]. Types supported by the system include reals, rationals, integers, strings, records, tuples, functions, tables, sets, and abstract data types. The type system also possesses two very powerful features, namely *dependent types* and *subtypes*, which are worth a closer look [41].

- *Dependent types, e.g., dependent record types:* A record type R having fields $f_1, \ldots, f_n$ of types $T_1, \ldots, T_n$ may be declared by R `: TYPE = [#` $f_1 : T_1, \ldots, f_n : T_n$ `#]`. Here, each type T_i may depend on the fields $f_1, \ldots, f_{i-1}$.
- *Subtypes:* If T is a type and P a predicate on T, then N `: TYPE = {x:`T `|` P`(x)}` declares a new type N whose elements belong to the largest subset of elements of type T that satisfy P.

Although subtypes and dependent types are very convenient to write specifications, they make type checking undecidable. PVS copes with this problem by generating *type correctness conditions* (TCCs), i.e., propositions that, when discharged by the user, guarantee type consistency. For example, when subtyping is used, a TCC is generated which states that the introduced subtype must not be empty, i.e., an object of the considered subtype must exist.

```
% Existence TCC generated for  N: {x:T | P(x)}
X_TCC1: OBLIGATION EXISTS (N: {x:T | P(x)}): TRUE;
```

In practice, TCCs can often be discharged automatically by using proof automation tools provided by the system. The rich type system enables one to encode *partial* functions, which are per default not supported in PVS, by restricting a function's domain appropriately using subtyping. Moreover, the type system also allows the sound support of recursive functions. More precisely, each recursive function declaration must include a *termination argument*. The type checker then generates a TCC stating that the termination argument is valid, i.e., it gives rise to a well-founded relation.

The proof checker [35] included in PVS is also interfaced to the Emacs editor. It constantly displays the current proof state in form of a *proof sequent*. Sequents may then be simplified by inputting *proof commands*, which may be considered as basic steps in the proof-construction process. PVS comes with a very rich set of proof commands that are concerned with equality reasoning, logical reasoning, and arithmetic reasoning. Basic proof commands deal with, e.g., skolemnization, case-splitting, or simplification. The system also provides a mechanism for writing *proof strategies*, i.e., proof scripts which are intended to increase the degree of automation within the theorem prover [35]. Strategies need to be written in Lisp and may use

pre-defined proof combinators, e.g., choice, sequencing, or iteration. Some interesting proof strategies are already delivered with PVS, including the strategy `tcc` specialized at discharging type correctness conditions and the very powerful strategy `grind` which combines logical and arithmetic simplifications in an intelligent manner. The proof checker also interfaces with decision procedures external to PVS, such as for temporal logics [15] and monadic second-order logic [17].

It is worth mentioning a feature which was introduced with the current release of PVS, namely the *ground evaluator* [36]. This component allows PVS ground expressions, i.e., executable definitions applied to concrete data, to be evaluated via compilation into Lisp. The efficiency of the obtained Lisp code crucially depends on the identification of situations of non-shared access to variables. This is done by static analysis techniques. Although ground evaluation is a first step towards animating specifications, the executable subset of the PVS language should be increased to handle certain kinds of symbolic evaluations in a future release of the tool.

Regarding tool customization, there exist currently two customization features in PVS other than the ability to provide user-defined proof strategies. One rudimentary feature uses *environment variables* for allowing a person to work with a text editor different from Emacs, for running the tool in batch mode, and for deciding which set of decision procedures to use. The other more powerful feature is the *library* concept which supports one in arranging theories and adding them to the system, e.g., integrating a theory for reasoning about graphs [9]. However, many key features in the current release of PVS are still fixed and cannot be customized. This especially concerns the PVS language, including its syntax and semantics, the type checker, and the type correctness conditions.

3. Past Experiences with Customization in PVS. In this section, we examine four academic and real-world case studies which we conducted in the past using PVS and which required us to employ PVS's customization features. Together the case studies cover most aspects of tool customization, including user-defined proof strategies, as well as syntactic and semantic issues of the PVS language.

3.1. Case Study: SAFER. The first case study, which involves the analysis of an embedded controller for NASA's Simplified Aid for EVA Rescue (SAFER), aims at exploring the limits of proof automation via semi-custom proof strategies. SAFER is a backpack propulsion system for free-floating astronauts, intended as a self-rescue device. It uses 24 gaseous-nitrogen thrusters to achieve six degree-of-freedom maneuvering control. Propulsion is available either on demand, i.e., in response to hand controller inputs, or through an automatic attitude hold (AAH) capability.

SAFER requirements were previously formalized using PVS during a NASA pilot project in formal methods, details of which appear in the appendix to a NASA guidebook [29]. In a nutshell, the SAFER system was specified as a state machine within PVS, and its properties of interest were encoded as system invariants which were proved by induction on the length of paths. A set of five property classes was identified, with matching proof schemes later devised. After refining the PVS proof strategies, fully automatic proofs of 42 model properties were obtained. Many properties were expressed as a "hold-until" formula, where an invariant holds over each sequence of states bracketed by the earliest occurrences of a trigger condition and a termination condition [12]. An example of the custom proof strategies is shown below, where "`state-tran`" is provided to prove basic state transition formulas.

```
(defstep state-tran (&optional (exp-fnums +) rewrites)
  (let ((auto-rewr (cons 'auto-rewrite (append rewrites (constant-rewrites)))))
    (then (skosimp*)
```

```
        auto-rewr (assert)
        (split-disjunctions*)
        (expand-rec-desc exp-fnums)
        (general-rewrites)
        (ground) (lift-if)
        (grind)))
  "(state-tran): prove state transition properties by expansion, replace & hide, grind."
  "~%Invoking state transition property strategy")
```

The sample property shown below is a direct expression of the following requirement: "*Once AAH is turned off for a rotational axis, it remains off until a new AAH cycle is initiated.*"

```
rot_axis_stays_off: LEMMA
  hold_until(
    LAMBDA s: toggle(AAH_state(s)) = AAH_on AND NOT active_axes(AAH_state(s))(r),
    LAMBDA s: NOT active_axes(AAH_state(s))(r),
    LAMBDA s: toggle(AAH_state(s)) = AAH_started,
    inputs)
```

Here, the three lambda expressions, parameterized with a state variable, specify the trigger, hold, and until-conditions, respectively.

Although custom proof strategies worked well in this case study, the PVS user community would benefit from greater insight into the prover's mechanisms. In order to implement more elaborate strategies that take function and lemma declarations into account, access to user theories would be needed. Since proof strategies, in our point of view, provide the key for making formal techniques attractive to engineers, they should be made as powerful and convenient as possible.

3.2. Case Study: Rewriting for User-defined Congruences. The second case study was devoted to developing rewriting support for user-defined congruences in PVS. The PVS language provides an *abstract datatype* mechanism [33] for defining new languages whose syntax is given in Backus-Naur Form (BNF). The semantics of a variety of languages, such as process algebras [5], is often defined via a behavioral congruence. While these congruences may be specified in PVS's higher order logic, the prover does not support rewriting for them. In fact, PVS's abilities for equational reasoning are limited to rewriting regarding the tool's built-in notion of equality. This is in contrast to many other theorem provers, such as HOL [16] and Isabelle [37], which provide a means for soundly introducing rewriting with respect to equations on user-defined congruences. In order to circumvent this shortcoming of PVS, researchers have embedded process-algebraic languages in PVS using *uninterpreted types* [4]. Although this approach opens the door for using the prover's built-in equality and its rewriting machinery, it forces one to sacrifice the most powerful proof principle supported by theorem provers, namely *structural induction.*

Our approach to the problem was based on providing a simple, customized, and conservative *proof strategy* for automating a single rewriting step with respect to a given congruence. In essence, our proof strategy, which is supposed to be applied to the sequent containing the term to be rewritten, only uses the *transitivity* property and the *compositionality* property of congruences. Especially, the strategy does not rely on uninterpreted PVS terms but works with PVS's abstract datatypes instead. The rewriting rule to be considered must be given as a lemma in PVS and serves as an argument to our strategy, which is defined as follows:

```
(defstep context-rewrite (equation)
  (then
    (use "transitivity")
      (hide 2)
      (use equation)
      (forward-chain "congruence")
      (hide -2)
      (inst -1 extract-context)
      (auto-rewrite "subst")
      (assert)
      (stop-rewrite "subst")
      (inst?)
      (assert)
      (hide -1))
 "Poor man's rewrite"
 "")
```

Here, `transitivity` and `congruence` are lemmas in PVS stating the transitivity and congruence property with respect to the considered congruence, respectively, and `subst` is a function which substitutes a context variable (a context is a term with a designated free variable) by a concrete term. Our strategy also requires the extraction of contexts in order to get to the subterm to which the rewrite rule `equation` should be applied. As with substitutions, extracting contexts is an easy exercise which can be performed by a function that is inductively defined along the structure of terms. Unfortunately, any specification of such a function needs to employ *syntactic equality* and, thus, cannot be implemented within the PVS language. Hence, the function `extract-context` for extracting contexts is directly defined in Lisp and uses a notion of syntactic equality which is defined internally within the PVS system. Although its definition is specific to the abstract datatype to which the considered term belongs, it can be automatically generated whenever an abstract datatype definition is introduced to PVS. However, the choice of a context may not be unique in the first place, since a rewrite rule may be instantiated in several ways with respect to a given term. Thus, either user guidance or the application of adequate heuristics is required. Since the techniques for term instantiation do not depend on the specific congruence under consideration, it should be possible to re-use the sophisticated pattern matching routines for PVS's built-in notion of equality, which are implemented as Lisp functions within the system.

Unfortunately, the poor documentation of the internals of the PVS system prohibited us from re-using existing routines for pattern matching and term instantiation. As final consequence, we did not meet our objective, namely to develop support for rewriting with respect to user-defined congruences in PVS.

3.3. Case Study: Integration of the B-Method. Although PVS is a rich tool for the analysis of formal specifications, it does not come with a built-in methodology for system development. In contrast, other tools include well-developed methodologies, such as the *B-method* [2], but provide very limited proof automation. Hence, the question arises whether, e.g., the B-method can be embedded in PVS in an elegant and cost-effective way.

The B-method is a state-oriented method which covers the complete life cycle of software development. It provides a uniform language, the *Abstract Machine Notation*, to specify, design, and implement software systems. A specification in B is composed of a set of modules which are referred to as *(abstract) machines.*

Syntactically, a machine consists of several clauses which determine its static and dynamic properties. For instance, the VARIABLE clause includes a set of variables that defines the state vector of the machine, the INVARIANT clause constrains the domain of states, and the OPERATIONS clause defines how states may be modified. The embedding of the B-method in PVS was done *structurally* [28], i.e., the expression language of B and the underlying logic of the abstract machine notation were encoded using the PVS language and the higher-order logic of PVS, respectively. More precisely, a front-end tool, called PBS [27], was implemented which supports the abstract machine notation in PVS. PBS works similar to a compiler as it takes an input file containing an abstract machine description and generates as output the corresponding embedding in the form of a PVS theory. This was necessary since PVS does not provide the possibility to extend the syntax of its specification language to accommodate the B-notation. When compiling a theory generated by PBS, the type checker of PVS generates type checking constraints. These correspond to proof obligations which assure soundness requirements of the machine under consideration, e.g., one proof obligation being that the machine's operations preserve the given invariant.

The semantic encoding of the B-method in PVS's higher-order logic maps machine states into a record type State, whose fields are the variables of the considered machine [6]. Machine invariants are introduced as a constraint predicate Invariant on State:

```
Invariant: [State → bool] = invariant
InvariantState: TYPE = {s:State | Invariant(s)}
```

where *invariant* is the invariant of the considered machine. Operations are then described by *generalized substitutions*, a semantic structure that includes a before-after relation between states, a pre-condition predicate, and a constraint which imposes the relation on states not violating the pre-condition. Generalized substitutions may be specified in PVS as follows:

```
Transition: TYPE = [# pre: [State → bool], rel: [[State,State] → bool] #]
Constraint: [Transition → bool] =
    LAMBDA (tr:Transition): (FORALL (e1,e2: State) (NOT tr`pre(e1)) ⇒ tr`rel(e1,e2))
GeneralizedSubstitution: TYPE = {tr:Transition | Constraint(tr)}
```

A transition between states is implemented as a record type Transition containing a pre-condition predicate pre and a before-after relation rel. Only transitions satisfying predicate Constraint are considered to be generalized substitutions, i.e., GeneralizedSubstitution: TYPE ={tr:Transition | Constraint(tr)}. As an example of a generalized substitution, consider the so-called *assignment substitution* $x_1, \ldots, x_n := e_1, \ldots, e_n$ which is encoded as

```
ASSIGN(f:[State → State]): GeneralizedSubstitution =
    (# pre := TRUE, rel := LAMBDA (e1,e2: State): e2 = f(e2) #)
```

where f is the function satisfying $f(x_i) = e_i$, for $1 \leq i \leq n$, and $f(x) = x$, otherwise. Please note that the encoding is constructed in a way that *soundness* of machines maps to *type correctness*.

Summarizing, although the B-machine syntax could not be integrated directly in the PVS language, the encoding of its semantics could be done conveniently. The latter is due to the expressive type system of PVS. In fact, the above encoding of machines makes use of subtypes and dependent types. However, type-system features absent in PVS could have simplified this task. As an example consider the situation where one machine imports other machines. Here, the state of the importing machine includes the states of the imported machine and their operations. This could have been elegantly expressed using *record subtyping*. Moreover, the proposal in [6] of a new operator for parallel substitution in the B-method suggests the utility

TABLE 3.1
Evaluation of the customization features used in the case studies

	Proof strategies	Language syntax	Language semantics
SAFER	good	n/a	n/a
Process algebra	poor	fair	poor
B-method	poor	fair	good
DDD	good	poor	poor

of general *polymorphism* in PVS. Last, but not least, it should be mentioned that specialized proof strategies for automatically discharging certain proof obligations, which arise during the translation of B-machines into PVS, were considered in [6]. Since the above-presented encoding strongly relies on PVS's type system, proof strategies would have required access to terms, types, and the type-checker. Although this access partially exists in form of internal Lisp functions of PVS, the lack of documentation let the attempt to write customized proof strategies fail.

3.4. Case Study: Support for the DDD-Method. In the fourth case study, we have studied how *Digital Design Derivation* (DDD) techniques, which provide a solid foundation for digital hardware design [20], can be combined and enhanced with the deductive capabilities of PVS [26].

Since design derivation semantics in hardware is based on mutually recursive *stream equations*, a PVS library defining a shallow embedding of stream equations was developed [18]. The stream library is modeled after PVS's abstract datatype mechanism, i.e., streams are encoded as an abstract co-datatype over an uninterpreted non-empty type constrained by axioms. The stream library provides support for co-recursive function definition and proof by co-induction. Strategies were developed in order to simplify the handling of proof obligations related to stream definitions and to partially automate proofs by co-induction. Moreover, multiple levels of interaction between design derivation and PVS were explored. The requirements for the design derivation are expressed in the PVS language. Algorithms satisfying the requirements are verified in the proof system of PVS and, then, are translated into a behavioral specification for a design derivation tool. Within such a tool, the behavioral description is refined into a concrete design. Refinements outside the scope of the derivation tool, e.g., regarding circuit optimizations that require sophisticated behavioral reasoning, are justified externally in PVS.

The approach described here has been illustrated by means of two significant examples, namely a *fault-tolerant clock-synchronization circuit* and an architecture for *floating-point division*. The examples show that PVS can be used effectively as a verification engine supporting DDD. However, limitations of PVS were experienced, both semantically and syntactically. From a semantic point of view, a shallow embedding reduces the potential to verify meta-results of the encoded theory. This is even worse in PVS, since the type system does not allow quantification over types. For instance, our embedding uses PVS tuples to represent tuples of the hardware design language, but properties concerning all tuple types cannot be expressed in our embedding. From a syntactic point of view, we were unable to create the desired syntactic forms for declaring *abstract co-datatypes*, since the PVS language cannot be extended. Thus, a significant portion of the development consisted of repeatedly creating declarations of lemmas in the specific form necessary for the correct operation of the strategies. A macro definition facility would have eased this task considerably.

3.5. Summary of Experiences. Table 3.1 summarizes and evaluates our experiences regarding the customization features of PVS, including proof strategies and the syntax and semantics of the specification language. The currently most useful feature to us is PVS's mechanism for custom proof strategies, although this mechanism would benefit from a better documentation and although it should provide (easier) access to the proof sequent and the prover itself. Regarding the PVS language, it is fair to say that although the language is very expressive, it does not support a means for customization and extensibility, syntactically as well as semantically. This is a major drawback for PVS, especially when compared to other theorem provers, such as HOL [16], Isabelle [37], and Coq [3]. All of them allow the user to modify and extend their specification languages, as needed.

Gaining theorem proving skills takes a large training investment. For formal specification and verification to enter common practice, high levels of proof automation are needed, as well as the ability to interface prover tools to languages and methodologies which are well-known to engineers. The lessons learned from the above case studies have demonstrated that issues of tool customization and extensibility are extremely important for the success of theorem provers in the engineering world, as well as for basic academic research in verification technology.

4. Enhancing Customization in PVS. Customization in PVS can be improved at different levels. In this section, we elaborate, component by component, on suggestions aiming towards a more customizable PVS. We hope that this contributes to the ongoing discussion within the PVS community about how to achieve greater customization and extensibility. The only component of PVS, we are completely happy with, is its Emacs interface. Emacs is widely known as a customizable and extensible editor which provides a nice integration of PVS with a variety of other applications ranging from authoring tools to Internet applications [43].

4.1. Specification language. As mentioned above, the specification language of PVS is essentially a strongly typed functional language enriched with operators taken from higher-order logics, such as quantifiers over general objects [34]. However, the language is not only targeted towards axiomatic and declarative specification styles, but it is expressive enough to encode other styles, including algebraic specifications [33], tabular specifications [31], and operational specifications [31].

The support for algebraic and tabular specifications in PVS, however, is particular. These kinds of extensions require modifications to the grammar of the PVS language, and a deep knowledge of the PVS internals, which is currently only available to developers at SRI International but not to the public research community. In view of the fixed grammar of the PVS language, the support of a new syntax usually requires the development of an *external parser*, such as PBS [27] in case of the above mentioned B-method. Additionally, superficial string manipulation may be wired at the Emacs level. Unfortunately, these solutions are far from optimal since they suggest the need for *decompilers*, which are known for being hard to develop. Alternatively, one may deal with compiler-generated encodings. Since such encodings hardly reflect the original specifications, their formal analyses are difficult. *Macro expansions* would be a simple way to mimic the syntax of external languages. In PVS they should be implemented in a way that the components of the system can refer back to the unexpanded language. Therefore, users would not have to change notation when switching back and forth between PVS and other tools.

Another challenging issue is the development of a mechanism to describe sub-languages of PVS. For instance, it might be helpful to restrict the specification language to consider only (i) finite types, in order to

be able to detect when, e.g., model checking [15] may be applicable, (ii) strategy constructs, when integrating the strategy language within the PVS language as will be discussed in Section 4.3, (iii) an executable fragment, e.g., for driving simulations, or (iv) a decidable theory, which may allow one to use more efficient decision procedures. An implementation of this mechanism via the type checker is suggested in the next section.

4.2. Type Checker. Extensions to the type checker of PVS, and in general to the type system, raise very delicate questions about the semantics of the system [35, 41]. For example, a naive extension of subtyping to consider *record subtyping*, such that fields may be added to records in the sense of object-oriented inheritance mechanisms, could render the system inconsistent. Indeed, the current set-theoretic semantics of PVS types, where subtypes have the meaning of subsets, is incompatible with most of the semantic approaches to inheritance in object-oriented languages [1, 24]. An intermediate solution to record inheritance in PVS is possible via the `CONVERSION` operator included in the PVS language. A *conversion* is a function that casts objects from one type to another. Conversions can be declared by the user, and the prover automatically uses them whenever necessary. However, as this latter mechanism is not controlled by the user, surprising errors may occur.

Polymorphism in PVS is limited to parameterization of theories, e.g., the theory "`generic[T:TYPE]: THEORY BEGIN ... END generic`" specifies a family of theories with respect to the abstract type `T`. However, the declaration "`polymorphism(T:TYPE):A = ...` " is not admitted by the system. We are not aware of the technical implications of general polymorphism in the type theory of the system, but this feature would be very handy in order to elegantly integrate other notations in PVS. In fact, semantic embeddings usually require meta-level encoding, for which polymorphism is a prerequisite.

Since constraining a theory is safe with respect to consistency, adding constraints to the PVS language might be useful to apply specialized algorithms on certain domains. For example, the type checker could be parameterized such that fragments of the language are recognized. Finally, the mechanism, which uses the type checker to automatically discharge type checking conditions, should be controllable by the user.

4.3. Proof Checker & Ground Evaluator. In order to write more powerful proof strategies, well-documented access to the proof context and the proof environment is needed. The proof context includes terms, types, sequents, theories, and type checking conditions, with currently only the access to proof sequents being documented [35]. Types contain information that can be helpful to decide which decision procedures are applicable in the considered proof situation. Access to theories is sometimes needed for looking up the availability of lemmas, and access to type checking conditions might allow one to automatically discharge them as they arise. The proof environment includes decision procedures, the theorem prover, and the type checker. Decision procedures need to be accessible if one wants to write, e.g., specialized versions of the strategy `grind`. In order to be able to interface PVS to external tools, the theorem prover itself must be accessible. Access to the type checker is needed when creating new PVS terms within proof strategies, such that new terms can be safely introduced to the prover. Finally, for the strategy language one might think of two advancements over the current speed-efficient standard which uses Lisp functions. A more elegant and still efficient solution would be to provide an *Application Programming Interface* (API), such that external languages can be used for writing strategies. Thus, the PVS prover would become controllable from external tools. The most elegant solution, which also addresses the question of soundness of proof strategies, would be to include the strategy language within the PVS language by introducing new language constructs and some pre-defined functions.

Regarding the support for rewriting with respect to user-defined congruences, user control of how the rewriting machinery and especially the pattern matching algorithm work is required. This may also help to avoid the problem of broken proofs when upgrading to newer versions of PVS. In order to achieve this goal, either documented access to the internals of PVS must be granted or, ideally, PVS needs to allow one the installation of rewrite rules parameterized with the specific congruence for which those rules are valid. Moreover, in the case of user-defined congruences, one also wishes to add and invoke new decision procedures with respect to the congruence of interest.

The ground evaluator introduced with PVS Version 2.3 should be considered an experimental feature, as its final functionality is still under discussion by its developers. Thus, it is too early to make detailed suggestions for enhancements. However, we hope that the evaluator will be extended to *symbolic* evaluation of specifications, i.e., those containing non-concrete data. This would enable the animation of a larger and more interesting class of specifications. The evaluator, ground or symbolic, should also be integrated with the other components of the system, namely the PVS language, the type-checker, and the proof-checker; right now, it is pretty much a stand-alone feature.

5. Conclusions. We conclude by suggesting several short-term and long-term goals for improving PVS's customization and extensibility, respectively. We hope that the PVS developers at SRI International will adopt some of these goals for future evolutions of their tool.

In the short run, a documentation of the PVS architecture as implemented in Lisp [44] using the Common Lisp Object System [10], of its interfaces, of its central classes, and of its objects should be provided. Moreover, an API for accessing the proof context and also the type checker within proof strategies should be developed. Together, this would enable all PVS users – not only those working at or visiting SRI International – to integrate various formal methodologies in PVS, e.g., for prototypically experimenting with *heterogeneous* verification techniques [40]. We believe, that the above suggestions will lead to a customizable PVS which allows for (i) increased automation via sophisticated proof strategies, (ii) tackling formalizations that are currently prohibited, and (iii) the division of labor for solving real-world verification problems. The latter point is especially important in practice, since it enables a more efficient conduct of projects by allowing a cleaner task separation between problem domain experts and PVS prover experts.

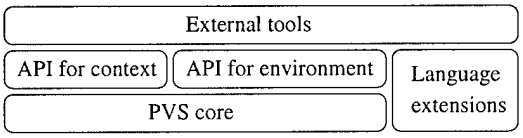

FIG. 5.1. *Architecture of an extensible PVS*

A truly extensible PVS, however, must not only be modular by allowing external tools to access the core of PVS via suitable APIs for both proof context and proof environment, but it must also support language extensions, syntactically as well as semantically. Each module customizing PVS's language must essentially extend the PVS core and must also provide an API in order for the new functionality to become available externally. An example architecture reflecting these ideas of extensibility is depicted in Figure 5.1. One may think of even more ambitious architectures, e.g., a more *federated* one, in which several tools can equally interact with each other. On top of the advantages of a modular PVS, an extensible verification system has the ability to take advantage of new theories and techniques as they occur in the field, as well as to merge the best of the many existing Formal Methods technologies.

Finally, we would like to mention that we are aware of the very delicate theoretical and technical issues behind our proposal. However, a more open PVS architecture is a prerequisite for research groups outside SRI International for being able to make more useful contributions to the PVS community. While the PVS language remains powerful and elegant, analysis techniques within the Formal Methods domain are becoming increasingly heterogeneous, drawing from a wide spectrum of specialized ideas. PVS would make an ideal flagship, but it cannot be the whole fleet. It would be wise to outfit PVS with a highly flexible means of cooperation, so that we all might enjoy a smoother cruise.

REFERENCES

[1] M. Abadi and L. Cardelli, *A Theory of Objects*, Springer-Verlag, 1996.

[2] J.-R. Abrial, *The B-Book—Assigning Programs to Meanings*, Cambridge University Press, 1996.

[3] B. Barras, S. Boutin, C. Cornes, J. Courant, J. Filliatre, E. Giménez, H. Herbelin, G. Huet, C. Muñoz, C. Murthy, C. Parent, C. Paulin, A. Saïbi, and B. Werner, *The Coq proof assistant reference manual: Version 6.1*, Tech. Report 0203, Institut National de Recherche en Informatique et en Automatique (INRIA), Rocquencourt, France, May 1997.

[4] T. Basten and J. Hooman, *Process algebra in PVS*, in Fifth International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99), R. Cleaveland, ed., vol. 1579 of Lecture Notes in Computer Science, Amsterdam, The Netherlands, March 1999, Springer-Verlag, pp. 270–284.

[5] J. Bergstra, A. Ponse, and S. Smolka, *Handbook of Process Algebra*, Elsevier Science, 2000. To appear.

[6] J.-P. Bodeveix, M. Filali, and C. Muñoz, *A formalization of the B-method in Coq and PVS*, in Electronic Proceedings of the B-User Group Meeting held in conjunction with the World Congress on Formal Methods (FM '99), Toulouse, France, September 1999, Springer-Verlag, pp. 33–49.

[7] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*, Object Technology Series, Addison Wesley Longman, 1998.

[8] R. Butler, J. Caldwell, V. Carreno, C. Holloway, P. Miner, and B. DiVito, *NASA Langley's research and technology transfer program in formal methods*, in Tenth Annual Conference on Computer Assurance (COMPASS '95), Gaithersburg, MD, USA, June 1995.

[9] R. Butler and J. Sjogren, *A PVS graph theory library*, NASA Technical Memorandum NASA/TM-1998-206923, NASA Langley Research Center, Hampton, VA, USA, February 1998.

[10] L. DeMichiel, *Overview: The Common Lisp Object System*, Lisp and Symbolic Computation, 1 (1989), pp. 227–244.

[11] B. Di Vito, *A formal model of partitioning for integrated modular avionics*, NASA Contractor Report NASA/CR-1998-208703, Vígyan Inc., Hampton, VA, USA, August 1998.

[12] ——, *High-automation proofs for properties of requirements models*, Software Tools for Technology Transfer, 2 (1999). To appear. Draft paper at http://shemesh.larc.nasa.gov/people/bld/sttt-bld.ps.

[13] B. Di Vito, R. Butler, and J. Caldwell, *High level design proof of a reliable computing platform*, Dependable Computing and Fault-Tolerant Systems, 2 (1992), pp. 279–306.

[14] B. Di Vito and L. Roberts, *Using formal methods to assist in the requirements analysis of the space shuttle GPS change request*, NASA Contractor Report NASA/CR-4752, Vígyan Inc., Hampton, VA, USA, August 1996.

[15] E. EMERSON, *Temporal and modal logic*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., vol. B, North-Holland, 1990, pp. 995–1072.

[16] M. GORDON AND T. MELHAM, *Introduction to HOL*, Cambridge University Press, 1993.

[17] J. GULMANN, J. JENSEN, M. JØRGENSEN, N. KLARLUND, T. RAUHE, AND A. SANDHOLM, *Mona: Monadic second-order logic in practice*, in First International Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '95), E. Brinksma, W. Cleaveland, K. Larsen, T. Margaria, and B. Steffen, eds., vol. 1019 of Lecture Notes in Computer Science, Aarhus, Denmark, May 1995, Springer-Verlag, pp. 58–73.

[18] U. HENSEL AND B. JACOBS, *Coalgebraic theories of sequences in PVS*, Journal of Logic and Computation, 9 (1999), pp. 463–500.

[19] *ICASE's Formal Methods Program.* Home page at http://www.icase.edu/~luettgen/icase.

[20] S. JOHNSON, *Synthesis of Digital Design from Recursion Equations*, The MIT Press, 1984. ACM Distinguished Dissertation.

[21] C. D. KLOOS AND P. T. BREUER, eds., *Formal Semantics for VHDL*, Kluwer, 1995.

[22] G. LÜTTGEN AND V. CARREÑO, *Analyzing mode confusion via model checking*, in Theoretical and Practical Aspects of SPIN Model Checking (SPIN '99), D. Dams, R. Gerth, S. Leue, and M. Massink, eds., vol. 1680 of Lecture Notes in Computer Science, Toulouse, France, September 1999, Springer-Verlag, pp. 120–135.

[23] P. MELLIAR-SMITH AND J. RUSHBY, *The Enhanced HDM system for specification and verification*, in Proceedings of the VerkShop III, Watsonville, CA, USA, February 1985, pp. 41–43. Published as ACM Software Engineering Notes, Vol. 10, No. 4, August 1985.

[24] B. MEYER, *Genericity versus inheritance*, ACM SIGPLAN Notices, 21 (1986), pp. 391–405.

[25] R. MILNER, M. TOFTE, AND R. HARPER, *The Definition of Standard ML*, MIT Press, 1991.

[26] P. MINER, *Hardware Verification using Coinductive Insertions*, PhD thesis, Indiana University, Bloomington, IN, USA, June 1998.

[27] C. MUÑOZ, *PBS: Support for the B-method in PVS*, Tech. Report SRI-CSL-99-01, Computer Science Laboratory, SRI International, Menlo Park, CA, USA, February 1999.

[28] C. MUÑOZ AND J. RUSHBY, *Structural embeddings: Mechanization with method*, in World Congress on Formal Methods (FM '99), J. Wing, J. Woodcock, and J. Davies, eds., vol. 1708 of Lecture Notes in Computer Science, Toulouse, France, September 1999, Springer-Verlag, pp. 452–471.

[29] NASA OFFICE OF SAFETY AND MISSION ASSURANCE, *Formal Methods Specification and Analysis Guidebook for the Verification of Software and Computer Systems, Volume II: A Practitioner's Companion*, Washington, DC, USA, May 1997. NASA-GB-001-97, Release 1.0. Available at http://eis.jpl.nasa.gov/quality/Formal_Methods.

[30] *NASA Langley's Formal Methods Program.* Home page at http://shemesh.larc.nasa.gov/fm.

[31] S. OWRE, J. RUSHBY, AND N. SHANKAR, *Analyzing tabular and state-transition requirements specifications in PVS*, NASA Contractor Report NASA/CR-97-201729, SRI International, Menlo Park, CA, USA, July 1997.

[32] S. OWRE, J. RUSHBY, N. SHANKAR, AND F. VON HENKE, *Formal verification for fault-tolerant systems: Prolegomena to the design of PVS*, IEEE Transactions on Software Engineering, 21 (1995), pp. 107–125.

[33] S. OWRE AND N. SHANKAR, *Abstract datatypes in PVS*, NASA Contractor Report NASA/CR-97-206264, SRI International, Menlo Park, CA, USA, November 1997.

[34] S. OWRE, N. SHANKAR, J. RUSHBY, AND D. STRINGER-CALVERT, *PVS Language Reference*, Computer Science Laboratory, SRI International, Menlo Park, CA, USA, September 1999. Available at http://pvs.csl.sri.com/manuals.html.

[35] ——, *PVS Prover Guide*, Computer Science Laboratory, SRI International, Menlo Park, CA, USA, September 1999. Available at http://pvs.csl.sri.com/manuals.html.

[36] ——, *PVS System Guide*, Computer Science Laboratory, SRI International, Menlo Park, CA, USA, September 1999. Available at http://pvs.csl.sri.com/manuals.html.

[37] L. PAULSON, *Isabelle: A Generic Theorem Prover*, vol. 828 of Lecture Notes in Computer Science, Springer-Verlag, 1994.

[38] *PVS bibliography.* Available at http://pvs.csl.sri.com/applications.html.

[39] *PVS specification and verification system.* Project page at http://pvs.csl.sri.com.

[40] J. RUSHBY, *Integrated formal verification: Using model checking with automated abstraction, invariant generation, and theorem proving*, in Theoretical and Practical Aspects of SPIN Model Checking (SPIN '99), D. Dams, R. Gerth, S. Leue, and M. Massink, eds., vol. 1680 of Lecture Notes in Computer Science, Toulouse, France, September 1999, Springer-Verlag, pp. 1–11.

[41] J. RUSHBY, S. OWRE, AND N. SHANKAR, *Subtypes for specifications: Predicate subtyping in PVS*, IEEE Transactions on Software Engineering, 24 (1998), pp. 709–720.

[42] J. RUSHBY, F. VON HENKE, AND S. OWRE, *An introduction to formal specification and verification using* EHDM, Tech. Report SRI-CSL-91-2, Computer Science Laboratory, SRI International, Menlo Park, CA, USA, February 1991.

[43] R. STALLMAN, *GNU Emacs Manual*, Free Software Foundation, 1997.

[44] G. STEELE JR., *Common Lisp: The Language*, Digital Press, Bedford, MA, USA, 1990.